

SPECIFICATION

Background of the Invention

This invention relates to physically non-distributed microprocessor based systems and, more particularly, the communication channel between functional components within a class of computer systems commonly known as personal computers.

A personal computer (PC) is comprised of several major functional components which may be basically defined as a microprocessor, a read/write memory (RAM), a mass storage device (e.g., hard drive or CD ROM), and an input/output (I/O) device (e.g., display, serial port, parallel ports, etc.). These functional components within the PC are interconnected by, and communicate via, a parallel data/address bus which is usually as wide as the processor data I/O path. The bus is typically of fixed physical length comprising a number of parallel copper traces on the PC's motherboard. In addition, there are provided a number of fixed tap points to the bus, e.g., edge connectors, din connectors, etc., to allow the customization of the PC's configuration by adding peripheral functions, memory, etc., or removing unused functionality.

While a bus provides a simple-minded mechanism for customization and communication within a PC, it has several limitations and unique problems associated with it. First, a bus is by nature, single transaction (e.g., only one functional unit

can communicate with another at any given time and during this time, no other functional units can communicate with anything) and sequential (messages follow one after the other with considerable handshaking between functional units). A second problem of a bus is that all functional units connected to the bus must meet the electrical specifications and requirements of the bus even if these specifications and requirements are quite dated, technologically. Thirdly, because the bus is a generic interconnect in nature, it can not be truly optimized for communication between any specific subset of functional units without adversely affecting communication performance between another subset of functional units.

Fourth, the speed of the bus is substantially slower than might otherwise be obtainable. This is due to two primary issues: First, busses are composed of relatively long lengths of parallel traces in close proximity to one another and this results in high parasitic capacitive coupling between traces of the bus (i.e., electrical noise). This noise increases as the frequency, or speed, of the bus increases. Thus, noise margin requirements restrict the speed (and length) of the bus. The second issue relates to the unknown and highly variable electrical loading of the bus. The speed of the bus is inversely proportional to the capacitive load on the bus. This capacitive load is determined by the number of electrical connectors on the

bus and the number of electrical connections to the bus. Since these numbers are variable, designers typically engineer the bus for worst case constraints. That is, the bus is typically slowed down to a rate that would sustain a worst case loading situation even though this may occur in one PC in a thousand.

Other major drawbacks of a bus are the need for electrical handshake signals and its fixed electrical data width (i.e., 8 bits, 16 bits, 32 bits, etc.) Handshake signals typically include READ, WRITE, MEM, I/O, WAIT/READY, etc. These signals are physical and are used to inform and control functional units (i.e., inform of the type of request, and control/synchronize between communicators.) Fixed data width limitations become problematic as chip data path widths exceed the width of the bus. As will be seen herein, defining handshaking and data size at the physical layer is less flexible than would be desired.

With the ever increasing demand for data manipulation in such applications as multimedia or graphics programs, the bottleneck of the bus becomes more acute. There have been many attempts to address and remedy this problem (e.g., VESA, Video local bus, PCI, etc.) but no solution offers greatly improved performance and complete scalability.

The present invention provides a system with the configuration flexibility of a bus-based PC while reducing the

electrical problems. Commensurably, interfunctional-unit communication speed and flexibility are greatly enhanced. The present invention applies a point-to-point packetized interconnection structure to facilitate communication between functional units (e.g., processor, memory, disk, I/O, etc.) within a PC.

Because it is point-to-point, the interconnections scheme of the present invention is of relatively fixed electrical load and can, therefore, be optimized for speed. Furthermore, the packet protocol that will be more fully disclosed herein provides a means of eliminating the typical physical layer control signals of a bus and replacing them with link-layer control which is much more flexible.

In order to allow for interconnecting more than two functional units, the present invention may be expanded by any of several interconnect topologies, e.g., switches, rings, etc. Where speed and a high degree of parallel traffic is desired, a switch topology provides the best means, e.g., crossbar switch. If speed is important but parallel traffic patterns are not very common, a shuffle-type switch may make the most sense. In applications that are very cost sensitive, the present invention may also be expanded by means of a ring topology.

As will be made clear in the specific disclosure portion of this document, the packetized point-to-point

interconnection scheme of the present invention improves speed and performance at reduced cost and with better noise characteristics (both internal electrical noise and radiated EMI) as compared to the bus interconnect currently employed within a PC.

Therefore, it is an object of the present invention to provide a new and improved PC, specifically, improving internal communication between microprocessor, memory, mass storage, I/O, etc. or any subset of these functional units. It is further an object of the invention to improve communication speed within a PC. It is further an object of the invention to reduce interconnection electrical noise within a PC. It is further an object of the invention to provide a more flexible interconnect means within a PC.

Accordingly, it is a general object of the present invention to provide a new and improved PC, specifically improving internal communication between microprocessor, memory, mass storage, I/O, etc. or any subset of these functional units.

It is a more specific object of the present invention to provide improved communication speed within a PC.

It is a still more specific object of the present invention to reduce interconnection electrical noise within a PC, and to provide a more flexible interconnect means.

Summary of the Invention

10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
110
115
120
125
130
135
140
145
150
155
160
165
170
175
180
185
190
195
200
205
210
215
220
225
230
235
240
245
250
255
260
265
270
275
280
285
290
295
300
305
310
315
320
325
330
335
340
345
350
355
360
365
370
375
380
385
390
395
400
405
410
415
420
425
430
435
440
445
450
455
460
465
470
475
480
485
490
495
500
505
510
515
520
525
530
535
540
545
550
555
560
565
570
575
580
585
590
595
600
605
610
615
620
625
630
635
640
645
650
655
660
665
670
675
680
685
690
695
700
705
710
715
720
725
730
735
740
745
750
755
760
765
770
775
780
785
790
795
800
805
810
815
820
825
830
835
840
845
850
855
860
865
870
875
880
885
890
895
900
905
910
915
920
925
930
935
940
945
950
955
960
965
970
975
980
985
990
995

The invention is directed to a physically non-distributed microprocessor-based computer system, comprising a microprocessor, a random access memory device, a mass storage device, an input-output port device, wherein the devices are each being operable in conjunction with the microprocessor and include an interface for receiving and transmitting data in packet form, and which further comprise a packet-based data channel extending between the microprocessor and the interfaces of the devices for providing simultaneous bi-directional communication between the microprocessor and the devices.

Brief Description of the Drawings

The features of the present invention which are believed to be novel are set forth with particularity in the appended claims. The invention, together with the further objects and advantages thereof, may best be understood by reference to the following description taken in conjunction with the accompanying drawings, in the several figures of which like reference numerals identify like elements, and in which:

Figure 1 is a block diagram of current PC systems which use a bus based interconnect.

Figure 2 is a conceptual block diagram of two packet nodes and a physical link.

Figure 3 is a general block diagram of a packet.

Figure 4 is a detailed diagram of the packet header

showing each of the fields.

Figure 5 is a general block diagram of an Idle packet.

Figure 6 is a general block diagram of an Request packet.

5 Figure 7 is a general block diagram of a Request Echo packet.

Figure 8 is a general block diagram of a Response packet.

Figure 9 is a functional block diagram of a link interface showing all the necessary elements.

Figures ^{and 10b} ~~10a~~ shows the possible structures of a request queue shown in Figure 9.

Figures ^{and 11b} ~~11a~~ shows the possible structures of a response queue shown in Figure 9.

Figure 12 shows the structure of a linc cache shown in Figure 9.

Figures 13a and 13b show possible implementations of the linc using discrete link interface chips.

Figure 14 shows a processor with an embedded link interface connected to system memory through another link interface.

Figure 15 shows detailed structure of the Idle packet for processor memory I/O.

Figure 16 shows detailed structure of the request

packet for processor memory I/O.

Figure 17 shows a detailed structure of a Request Echo packet for processor memory I/O.

Figure 18 shows detailed structure of a Response packet for processor memory I/O.

Figure 19 shows sample request packets for a load and store instruction.

Figure 20 is a detailed schematic of the processor node interface chip transmit half.

Figure 21 is a detailed schematic of the processor node interface chip receive half.

Figure 22 is a detailed schematic of the memory node interface chip receive half.

Figure 23 is a detailed schematic of the memory node interface chip transmit half.

Figure 24 shows a possible ring interconnect topology for packet nodes.

Figure 25 shows a possible switched interconnect topology for packet nodes.

Figure 26 shows possible structures of response packets for a given request in response to conditional branch or jump in program code.

Figure 27 is a flow graph of the processor node receive protocol.

Figure 28 is a flow graph of the processor node transmit protocol.

Figure 29 is a flow graph of memory node receive protocol.

5 Figure 30 is a flow graph of memory node transmit protocol.

Description of the Preferred Embodiment

With reference to the Figures, and particularly to Figure 1, a node is defined as any device or group of devices which perform a specific system function within a microprocessor system. Such nodes typically have a physical address or addresses with which they are associated. Examples of such nodes include a processor, a memory, an input/output device, a hard disk, etc. The object of this invention is to interconnect all, or some subset of all, nodes with a high speed message passing packetized interconnection channel.

Figure 1 depicts the simplest implementation of the prior art. Processor node 40 is connected to bridge node 41. Bridge node 41 is further connected to memory node 42. Bridge node 41 also provides an interface to system bus 43 in order to allow processor node 40 to communicate with peripheral nodes 44, 45, 46 and 47 over the bus.

The high speed message passing packetized interconnect channel and protocol is intended to allow asynchronous

communication between any two nodes, so equipped, in any system configuration using packets of data or instructions. Figure 2 shows a logical conceptualization of a node. The functional unit 48 may be any of the elements of Figure 1. Interface 49, which is the embodiment of the present invention, provides a seamless interface between functional unit 48 and the physical layer 50. Physical layer 50 connects to any other packet node 51, thereby, in conjunction with interface 49, allowing communication between any group of functional units of Figure 1.

Thus, the present invention makes each node in a computer microprocessor/microcontroller based system interface with each other using a uniform packed based message passing interface. The invention creates packet nodes which speak the same language electrically and logically. This simplifies communication and minimizes traditional handshaking and overhead. Therefore, significant data-rate speedup may be gained because rather than being pigeonholed into a hardware restricted, single transaction, high overhead interconnect (i.e., a bus), the present invention allows messages of varying length (rather than single transaction) with flexible handshaking, and minimal overhead. Or, more precisely, the present invention allows for more intelligent and flexible information exchange between nodes while requiring only that overhead and handshaking required for a specific transaction. The present invention does not impose, as

does a bus, a set of costly rules and formalism that must be adhered to even if it makes no sense for a specific transaction.

Again, with reference to Figure 1, current systems communicate at the hardware level, i.e., the processor node 40 issues commands on the processor bus which must then be converted from the virtual address that the processor understands to a physical address by bridge 41. This, information is then further converted by bridge 41 prior to being placed on the bus 43. At this point, all other nodes, 42, 44, 45, 46 or 47 on the bus look to see if they are being addressed by processor node 40. The one node that is in fact being addressed, acknowledges the processor query and then takes action based on that query. Upon receiving the acknowledgment from the queried node, processor node 40 now may take appropriate data action (i.e., output data, input data, etc.) and the transaction is assumed to be completed. Next, the process can repeat in exactly the same manner, even if the query is to the exact same node as previously addressed. This is the handshaking and overhead bottleneck of a conventional bus.

Furthermore, whenever the bus is being utilized by a pair of nodes, all other nodes are prevented from communicating with any other node. The present invention stems from the realization that the byte by byte handshaking and transmission that is typical of a bus-based system is greatly inefficient and constraining and may be vastly improved upon.

In accordance with the invention packet based message passing techniques are applied to the nodes within a microprocessor based system. With reference to Figure 2, the following is a simple illustration of the invention's operation:

5 Execution unit 48a (e.g., processor node) requests information from execution unit 48(b) (e.g., memory node) via link interface 49(a). Link interface 49(a) assembles a packet requesting said information (i.e., its address in memory, amount of data requested, etc.) and then rapidly transmits said packet to link interface 49(b) via physical interface 50 (which may be single-ended line drivers, low voltage differential drivers, or any other method common in the art). Link interface 49(b) then decodes the packet and takes the necessary steps to process the request with respect to execution unit 48(b). Link interface 49(b) then collects, from execution unit 48(b), all data necessary to fill the request, packetizes the data into a response, and then, when the request is filled, ships the data back to link interface 49(a) via physical link 50. At this point, link interface 49(a), depacketizes the data and provides
20 it to execution unit 48(a) in a manner befitting the execution unit's request. Thus, the physical link is only tied up for the time when useful information is actually being sent. Furthermore, requests for several pieces of data result in less physical interface bandwidth utilization since the several pieces

of data are streamed in the same message.

A packet node in the present invention communicates with another packet node using "packets." Each packet contains all the necessary information that is required for the intended receiver without the added overhead of setting up the receiver or formatting the data to a node-specific set up.

The structure of a packet 55 such as shown in Figure 3 may have, for example, the following general characteristics:

- 1) Packet components 52 and 53 are of 16 bits (2 bytes) in width and referred to as a packet word.
- 2) Header 52 has means for indicating that the next packet word is an extension of the header 52. This means is the extended header bit 54.
- 3) A packet body 53 which can be anywhere from 0 bytes to 256 bytes in length.
- 4) The maximum size of a packet 55 is 258 bytes.
- 5) The width of the packet remains the same regardless of the width of the channel (i.e., for wider channels, more than one packet word may be sent in parallel.)

The above definitions may easily be changed without affecting the nature of the invention.

Figure 4 details the fields within the header packet. The type-of-packet field 59 defines one of four fundamental

packet types that are exchanged between any two nodes. These are
1) IDLE, 2) REQUEST, 3) REQUEST ECHO, and 4) RESPONSE. The type-
of-instruction field 58 indicates the action that needs to be
taken by the receiving node of this packet. Examples of such
actions include load, store, input, output, read, write, and
other system level interfunctional unit operations.

The size-of-device field 57 is to allow for the
interface between devices of different physical data widths with
minimum physical layer transmission time. By knowing the size of
a requesting node device, the interface circuitry of the
receiving node can pack the data into a packet in the most
efficient manner for decoding by the requesting device and only
send portions of the overall required response that are filled,
where 'filled' is defined as sufficient to meet the width of the
requesting node device as defined in the size-of-device field.

The flow control field 56 contains the size-of-
response, node ID, extended header bit, and BUSY/OKAY status bit.
The size-of-response indicates the amount of data being
requested. The node ID indicates the logical functional unit for
which this packet is intended. The extended header bit allows for
headers greater than 2 bytes in size, where necessary. The
BUSY/OKAY status bit indicates whether the receiver of request
packet can accept and service the packet.

Having defined the fields of the header packet, we now

define the four fundamental packet types. The IDLE packet 55a, Figure 5, contains only the header and no data and is continually sent out by the idle node. A node receiving the IDLE packet may then use the idle link for transmitting data that the receiving node believes the idle node may need based on the idle node's prior request history.

It is informative to illustrate the use of IDLE packets with an example. Assume that the processor node has been requesting sequential data blocks from the memory node. At some point in time, the processor node stops requesting data from the memory node because the processor node has to do something else (e.g., service an interrupt). At this point, the processor node sends idle packets to the memory node. Upon reception of the idle packet, the memory node reviews the history of processor node requests and may continue to send data based on a projection of the history of the processor node requests. These unrequested data are then stored in the processor node link interface cache provided the processor node has not specifically requested some data from any other node. In this way, idle links can be used most effectively to transmit data that may be needed before it is requested.

A second type of packet is the REQUEST packet 55b, Figure 6, which has been informally referred to throughout this disclosure. This packet is transmitted between any two nodes to

indicate or request an action from the receiving node for the requesting node. The request packet contains a header (see Fig 3, element 52) that has the ID of the requested node in the flow control field and the type of instruction for the receiver to execute, e.g., load, store, etc. The request packet also contains a body (see Fig. 3, element 53) to the extent that there is data sent by the requesting node to the receiving node for the receiver to perform the requested instruction.

The REQUEST ECHO packet 55c, Figure 7, is sent by the receiving node to acknowledge reception of a REQUEST packet 55b. This packet is primarily for indicating whether the request from the requester can be catered or not. Within the header of the request echo packet, in the flow control field, the REQUEST ECHO packet indicate whether the receiver is busy (busy echo) or able to service the request (okay echo).

The last packet type is the RESPONSE packet 55d, Figure 8. This packet is used to respond to a REQUEST packet. The header of the RESPONSE packet contains the node ID of the intended receiver (i.e., the original requesting node) and other information regarding flow control, etc. The body of the response packet contains the data requested to the extent it is required and the body of the RESPONSE packet is no longer than it needs to be to hold said data.

A typical transaction between any two nodes (node a

and node b) shown in Figure 2 is summarized below:

1. Node A generates a request packet for Node B.
2. Node B, based on whether Node B's request queue can cater to the request, sends one of two messages back.:
 - a. If it can cater to request from Node A then sends Node B a REQUEST ECHO OKAY packet.
 - b. If it can not cater to a request from Node A then Node B sends a REQUEST ECHO BUSY packet.
3. If Node A receives a REQUEST ECHO OKAY packet then Node A takes no action on the original request. If Node A receives a REQUEST ECHO BUSY packet then Node A resends the original request.
4. If REQUEST ECHO OKAY was sent by Node B, then Node B sends a RESPONSE packet to cater to the original request. This completes the transaction between Node A and Node B.

If no action is required from either Node A by Node B or from Node B by Node A , then IDLE packets are exchanged between them. Node A sending an IDLE packet to Node B or vice versa are both independent operations. The IDLE packet may also be used to exchange configuration / status / control information of each node.

To extend the capability of the present invention to an

arbitrary number of nodes requires an expanded interconnect. Figure 24 and Figure 25 depict two possible interconnection schemes.

Figure 24 shows a topology commonly referred to as a ring interconnect 67. In this type of interconnect, each link interface's physical link output is connected to a neighboring nodes link interface physical input until the ring is closed. For this type of implementation, the link interface must implement a pass-through mechanism. That is, each link interface must compare its node ID to the node ID of the packet header. If the compare is not successful, the link interface must forward the packet just received on its physical link input to its physical link output. In this way, packets circulate in the ring until they arrive at their ultimate destination. This receive-check-forward mechanism is functionally similar to that described in the IEEE 1596 (SCI) specification (Elastic Buffer).

To improve performance over a ring, Figure 25 shows another common interconnect topology commonly referred to as a switch 68. The switch 68 of Figure 25 may be a crossbar switch, a shuffle switch, a broadcast crossbar switch, or similar device. Implementations of crossbar switches are well known to the art and it is sufficient to describe a cross bar as N, M to 1 multiplexers, where N is the number of output ports and M is the number of input ports. When a crossbar switch is used, each link

interface must check the node ID of the received packet to guarantee that the packet is intended for the receiving node. This straightforward modification to the link interface physical link input circuitry is to include an ID decoder in the receive logic before queuing the request.

Figure 9 depicts a functional diagram of the link interface 49. The Host Interface 60 provides the means to connect the link interface to the bus of the functional unit 48 (i.e., processor, memory, I/O, disk, etc.) This part of the link interface contains all the necessary hardware to handshake with the functional unit node and is specific to the said functional unit. It also provides for all necessary signals to complete bus cycles needed for the functional unit.

The Store Accumulator 61 is responsible for packing data into a packet body for the STORE instruction. This is especially useful when the processor node is doing a burst write. In this case, the several data and addresses that are sequentially output by the processor are accumulated by the Store Accumulator 61 of the link interface and packed into one store message packet. Thus, a single message transaction results in several data being stored by the receiving node.

The Control block 62 provides for control of the internal components of Figure 9 as well as coordinating the functioning of the physical link. Control block 62 is

essentially a state machine that keeps track of the link state and provides the necessary housekeeping functions of the link. Control block 62 also contains the history register which is used by the requester in conjunction with the Linc Cache 63 'hit' information to determine the desired size-of-response for a given request. This same register is used by the receiver to determine how much data and from where said data may be returned when the receiver detects an IDLE packet. The detailed operations that Control block 62 performs are disclosed in association with the operation of each the blocks of Figure 9 and the link interface.

Request Queue 64 provides buffering and storage for all accesses coming from a functional unit 48. These accesses can either be stored in raw form (node address and data format), or in packetized form, depending upon the access arrival rate. That is, based on the rate that accesses come into the link interface from the functional unit 48, the access may be stored raw and then packetized as the access is converted to a request packet and placed on the physical link or the access may be packetized prior to being placed in the queue. With reference to Figure 10, if the access is being stored as a packet in the Request Queue 64, the queue is configured, as in Figure 10a, to be two bytes wide plus one bit for frame. The frame bit 67 is used to indicate the presence of header on the current cycle. If the access is being stored raw, then the request queue is

reconfigured to be as wide as the functional unit's address width 68 plus data width 69 plus the instruction field width 70 as in Figure 10b.

5 The Echo Waiting Queue 65 of Figure 9, is operable to function as storage for outstanding Request packets. These Request packets are copied into the Echo Waiting Queue 65 until an ECHO OKAY packet is received from the node catering to the request. Storing outstanding requests provides the means for the Control Block to handle out-of-order RESPONSE packets and to verify link integrity by making sure that all requests are being responded to.

10 The Response Queue 66 has a structure similar to the Request Queue 64. It may be configured as in Figure 11 and its operation is the reverse of the Request Queue. The Response Queue can store either packetized information as received from the physical link for later depacketization and passage to the functional unit (function unit data-need is slow), Figure 11a, or information received from the physical link may be immediately depacketized and stored raw for delivery to the functional unit (functional unit data-need is fast), as in Figure 11b.

20 Linc Cache 63 is basically a directed mapped cache for caching response data for the functional unit. The size of the Linc Cache 63 is an integer multiple of the maximum data packet size 53, i.e., $m \times 256$. To keep track of the latest data, the

Linc Cache 63 is partitioned into two identical blocks; one block containing the latest information 71 and the other block containing the information received before and up to the latest update 72.

5 With reference to Figure 12, each block of the Linc Cache 63 is of fixed size (width 73 and depth 74). However, the Linc Cache line 75 size is variable. Furthermore, the Linc Cache line size is always at least as long as the functional unit's cache line 76 size. There are N words in a Linc Cache line, where N is dynamically variable. If an access from the functional unit misses in the Linc Cache, the link interface will request the data from the proper node. The amount of data requested by the link interface, N, depends on the history of the 'hit' rate within the Linc Cache. If the hit rate is high, the control circuitry increase the Linc Cache line size thus maximizing data transmission per physical link transaction. If the hit rate is low in the Linc Cache, the Linc Cache line size is reduced in order to reduce the size of messages on the physical link.

20 The motivation for this unique and counterintuitive approach to cache management is the realization that if the hit rate to the Linc Cache is low, the accesses are almost certainly not sequential and are unpredictable. Thus, increasing the Link Cache line size will probably not improve the hit rate.

Therefore, the invention reduces the line size (which will probably not hurt the hit rate but will make the link available for all nodes more often since message sizes from this node will now be smaller).

5 Figures 13 and 14 depict two possible embodiments of the link interface 49. Figure 13 shows an implementation wherein the link interface is separate from any of the node's circuitry. Figure 13a shows an implementation where the physical link is on a motherboard or external physical channel with link chips for each node. The link chips in Figure 13a only contain the link interface 49. Figure 13b shows an implementation where the physical link is embedded inside a single chip or a linc chip 39. The linc chip 39 contains both a link interface 49 and the physical link or channel 50. Figure 14 shows an implementation wherein the link interface associated with the processor node is include within the processor silicon itself. Figure 13b will be discussed first.

10
15
20 Figures 20 through 23 show a detailed implementation of a link interface. With no loss of generality, the interconnect is assumed to be a two point, point to point interconnection between a processor node and a memory node. It will be shown in the description to follow how the memory node interface may be extended to include any peripheral device

 Figures 20 and 21 depict the transmit and receive,

respectively, link interface circuitry for the processor node. With respect to Figure 20, the transmit operation starts when the processor node (host) begins a bus cycle. The transmit circuitry latches the necessary data and address information into latch 201. As the data and address are being latched, the bus cycle of the processor node is decoded to be either a read or write (load or store). In either case, control circuitry 204 enables the appropriate header from header pool 207. This header pool encodes all possible header types since the type of processor is known to the specific implementation of the link interface and allows for faster assembly of a packet. The header is put in the request queue 202 and then the address/data/control of the bus cycle is mapped to the packet as it is put in the request queue 202. The request queue 202 operates in a first in, first out (FIFO) manner. If the cycle is deciphered to be a store then the request is stored in the store accumulator 205. After the store accumulator 205 is full the control attaches a header to the information in the store accumulator 208 and sends it out on the physical linc 50. Once any request has been sent out, it is queued in the echo waiting queue 203 to await the receipt of an ECHO OKAY. During the queuing of a request, when a load or read request is received from the processor node, there is a search done in the linc cache 206 and the store accumulator 205 to determine whether the request can be catered to without going out

on the linc (i.e., check to see if a linc cache hit occurs).

Referring to Figure 21, in the receive operation, once a packet is received over the physical link, the intelligent latch 208 and demultiplexer 209 combination allows

5 depacketization of the incoming information and the storing of the data/address in the linc cache 206. Because of the variable linc cache line size, the returning requested data will be more than the original processor required amount (to fill the larger linc cache line). Finally, the control 204 issues the appropriate bus and control signals to satisfy the processor node request.

The receive operation for the memory node with respect to Figure 22 is described below. The data comes in on physical link 50 and latch 208 takes the data and appropriately fills the request queue 210. The request queue 210 contains the raw request which then goes to the memory controller 211 for the appropriate action. The design and implementation of the memory controller is specific to memory devices being used and the system memory architecture. Once a load request has been
20 received, the controller 209 checks the linc cache 212 to see if the data is ready to be packetized and sent across the physical link 50. This saves cycles since it is not necessary to assert the appropriate control signals to start the memory access cycle if there is a hit in the linc cache 212. If the request received

is a store request, then it is directly sent to the memory controller 211 for appropriate action.

The transmit operation with respect to Figure 23 is briefly described below. Once a request is received, header pool 213 is indexed to provide the appropriate header for the response packet. The request that was received over physical link 50 and stored in the request queue 210 is dequeued and the response is then taken out of the link cache 212 or the memory and packetized by attaching the appropriate headers and put into the response queue 214. The response queue has a FIFO operation similar to any of the other queues used in the current implementation and so the response will be sent when it is at the head of the queue.

It is important to realize that Figures 22 and 23 describe an Functional Unit specific implementation embodying a memory node. With the addition of a store accumulator and the appropriate controller that replaces the memory controller 211 in Figure 22, this circuitry can be adapted to Functional Units of any type on any node.

The protocols used for communication between the processor node and the memory node are described in the flow graphs in Figures 27, 28, 29 and 30. These protocols are:

- 1) Processor node Protocols
 - a) Receive Protocol (Figure 27)
 - b) Transmit Protocol (Figure 28) and

2) Memory node Protocols

- a) Receive Protocol (Figure 29)
- b) Transmit Protocol (Figure 30)

The flow graphs in conjunction with the description described herein describe the functioning of the link between the processor node and the memory node.

Figure 14 depicts a provision of the link interface on the same silicon as the processor. This results in several important simplifications and improvements. Moving inside the processor silicon allows the link interface access to the Translation Lookaside Buffer (TLB) and the Branch Target Buffer (BTB) which allows the implementation of sophisticated prefetching and caching schemes.

Microprocessors typically do speculative execution based on the load/store instructions in a program. For standard arithmetic logic unit (ALU) operations, it is relatively easy to identify the register operands needed to be accessed during the instruction decode phase itself. However, for memory access operations, significant improvement is possible. In particular, the determination of the memory location that needs to be accessed requires an address calculation. The load/store instructions are issued to a pre-execute engine where address calculation is performed. After address calculation, the virtual address is translated into a physical address, if necessary.

This address is then issued to the memory interface of the processor to form the appropriate request.

By moving the link interface inside the processor, access is gained to TLB which stores a lookup table or cache translation descriptors of recently accessed pages. This information is very valuable in doing intelligent memory prefetches because now the link interface can look at the TLB and decide the location and access size of prefetches.

When the link interface is within the processor, the interface also gains access to the BTB. This allows two important benefits. First, in any given program or code, there typically is a branch or jump every five instructions, on the average. The branch prediction mechanism of the processor allows the processor to do speculative execution by predicting where it needs to go four or five instructions ahead of the current program counter (PC). When the branch predictor is wrong, there is a huge performance penalty in the processor because of stalls and pipelines running empty. By having access to the BTB, the link interface knows all possible outcomes of the branch and can prefetch data/instructions for all of the possible outcomes and have this information available at the processor. In this way, performance penalties due to branch prediction errors are significantly reduced. This implements a virtual zero wait-state operation to memory on a branch miss.

Second, in the rare event that there are no jumps in the program, Figure 26 shows a method of obtaining performance improvement. Rather than fetch a large block of data from a single address, each packet would be configured to fetch several smaller blocks of data from several different addresses. In this context, "data" means both program data and processor instructions.

For all of the above detailed embodiments of the invention, Figures 15, 16, 17, 18 and 19 show the specific implementations of the packet structures along with which fields of the header are active for each packet type.

While a particular embodiment of the invention has been shown and described, it will be obvious to those skilled in the art that changes and modifications may be made therein without departing from the invention in its broader aspects, and, therefore, the aim in the appended claims is to cover all such changes and modifications as fall within the true spirit and scope of the invention.